Dynamic Cache Partitioning for Multi-core Systems

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering

By

Yang Zhang
B.S., Shenyang Normal University, 2011

2014
Wright State University

Meilin Liu, Ph. D.
Thesis Director

Mateen Rizki, Ph. D.
Department Chair

Committee on
Final Examination

Meilin Liu, Ph. D.

Jack Jean, Ph. D.

Soon Chung, Ph. D.

Robert Fyffe, Ph. D.
Dean, School of Graduate Studies
ABSTRACT


As the power consumption (power wall) is limiting the clock frequency increase, multi-core and many-core processors become the major-trend of the new generation of processors. One of the biggest challenges to achieve high performance in multi-core systems is the growing disparity between processor and memory speeds. The “memory wall” problem, i.e., the growing disparity of speed between the processor and the memory, becomes even more serious in the multi-core systems.

Caches have been highly successful in bridging the processor-memory performance gap by providing fast access to frequently used data. Caches also save power by limiting expensive off-chip memory accesses. In this thesis, we propose a Dynamic Cache Partitioning Scheme to improve both temporal locality and spatial locality and to take advantage of the memory hierarchy in the multi-core systems. We use multi2Sim, a multi-core simulator, to collect the memory traces for the test benchmarks. The collected memory traces were used as the test inputs for the proposed Dynamic Cache partitioning Scheme. The simulation results showed that the cache misses were reduced significantly by the proposed Dynamic Cache Partitioning Scheme.
# TABLE OF CONTENTS

CHAPTER 1.  INTRODUCTION ........................................................................................................... 1
  1.1 Background and Motivation ................................................................................................. 1
  1.2 Problem Statement ............................................................................................................. 2
  1.3 Thesis Organization .......................................................................................................... 4

CHAPTER 2.  BASIC CONCEPTS ABOUT CACHE PARTITIONING ............................................ 6
  2.1 Introduction .......................................................................................................................... 5
  2.2 Architecture Overview ....................................................................................................... 5
  2.3 Memory Hierarchy ............................................................................................................. 7
  2.4 Cache Organization .......................................................................................................... 10
    2.4.1 Cache Hierarchy ........................................................................................................... 11
    2.4.2 Cache Entries ............................................................................................................ 12
    2.4.3 Cache Associativity .................................................................................................. 14
  2.5 Cache Partitioning .......................................................................................................... 16
    2.5.1 Hash Functions ........................................................................................................ 17
    2.5.2 Replacement Policy ................................................................................................... 18
    2.5.3 Bloom Filter ............................................................................................................ 19

CHAPTER 3.  DYNAMIC CACHE PARTITIONING FOR MULTI-CORE SYSTEMS .................... 21
  3.1 Introduction ....................................................................................................................... 21
  3.2 Overview ........................................................................................................................... 21
  3.3 Tracing Cache Misses ........................................................................................................ 25
  3.4 Partitioning Algorithm ..................................................................................................... 26
  3.5 The Exploration of The Unassigned Area Scheme ............................................................ 30
    3.5.1 Overview ................................................................................................................ 31
    3.5.2 The Unassigned Area Cache Partitioning Algorithm ............................................. 32
CHAPTER 4. SIMULATION ............................................................................................... 35
  4.1 Introduction ........................................................................................................... 35
  4.2 Simulation Environment ....................................................................................... 35
  4.3 Installation of Multi2Sim ..................................................................................... 38
  4.4 Memory Trace Collection .................................................................................... 39
  4.5 Experimental Setup ............................................................................................. 41
  4.6 Simulation Results and Analysis ........................................................................ 43
CHAPTER 5. RELATED WORK .......................................................................................... 49
  5.1 Introduction ........................................................................................................... 49
  5.2 Replacement Policies ........................................................................................... 49
  5.3 Degrading The Miss Rate .................................................................................... 50
  5.4 Low Energy Cache ............................................................................................. 51
  5.5 Other Approaches ............................................................................................... 52
CHAPTER 6. CONCLUSIONS AND FUTURE WORK ..................................................... 53
REFERENCES ............................................................................................................... 55
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Architecture Overview of an Uni-core Processor</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>The Architecture Overview of a Multi-core Processor</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Memory Pyramid</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Mapping Schemes for Three Cache Type</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Bloom Filter Principle</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview of System Structure</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Cache Scheme Components</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Overview of the Unsigned Area Cache Partitioning Scheme</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Structure of Multi2Sim Source code</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>The Architecture Phases of the Multi2Sim</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Results of Mix01 by Our Cache Partitioning Scheme in a Quad-core System</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Results of Mix02 by Our Cache Partitioning Scheme in a Quad-core System</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Results of Mix03 by Our Cache Partitioning Scheme in a Quad-core System</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Results of Mix04 by Our Cache Partitioning Scheme in a Quad-core System</td>
<td>48</td>
</tr>
</tbody>
</table>
LIST OF TABLES

2.1 Effects on Miss Types for Increasing Cache Parameters ........................................ 14
4.1 Parameters of The Cache Simulator Configuration.................................................. 42
4.2 Combinations of Benchmarks.................................................................................. 43
ACKNOWLEDGEMENTS

I would like to express my special appreciation to my thesis advisor, Dr. Meilin Liu, for her continuous guidance and encouragement throughout my thesis research and the writing of the thesis.

I would like to express my warm thanks to Drs. Jack Jean and Soon Chung, my thesis committee members for the precious advice to improve the document.

Then, I would like to thank Lifeng Liu for sharing the valuable and truthful views for various problems related to the thesis.

I also would like to have this opportunity to thank my family. I can’t even complete my thesis research without their assistance and encouragement.
CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

As the power consumption (power wall) is limiting the clock frequency increase, multi-core and many-core processors become the major-trend of the new generation of processors. One of the biggest challenges to achieve high performance in multi-core systems is the growing disparity between processor and memory speeds.

Memory performance has a dominating effect on overall system performance, especially for the data-dominated applications [15,17,19]. Also, memory usually occupies half of the surface of the integrated chip of multi-core system; the total amount of memory required is also a main contributor of the manufacturing cost and the chip size of the multi-core CPU. Memory will also decide the power consumption, since memories and buses consume large quantities of energy. The “memory wall” problem, i.e., the growing disparity of speed between CPU and memory, becomes even more serious when throughput in the processor part is propelled by multi-core architectures.

Cache works as temporary buffers for data, and it is faster but smaller than the off-chip main memory. Caches have been highly successful in bridging the processor-memory performance gap by providing fast access to frequently used data. Caches also save power by limiting expensive off-chip memory accesses. The IBM POWER5 processor gives us a good
Example. In this case, 280 cycles are required when accessing the data located in the off-chip main memory, and accessing the on-chip level two cache just requires 14 processor clock cycles [5].

However, the efficiency of caches has been limited due to energy inefficient tag checking and comparison logic [10]. Previous work adopts both static and run-time approaches to distribute data in the cache to improve the cache performance and correspondingly the system performance. Page coloring [6] is also a well-known technique that reduces conflict misses by mapping consecutive pages in the virtual address space into consecutive cache frames.

Many architectures such as PowerPC attempt to avoid the need for cache line locking through the use of cache control instructions [4]. Marking an area of memory with “Data Cache Block Zero” (allocating a line but setting its contents to zero instead of loading from main memory) and discarding it after use (“Data Cache Block Invalidate”, signaling that main memory needn’t receive any updated data), the cache is made to behave as a scratch pad memory. Different granularity of cache partitioning, set-alignment [15], way-alignment [16] and even line-alignment [17] have also been proposed to improve system performance.

1.2 Problem Statement

In the multi-core systems, sharing the cache among several cores brings new constraints and challenges to cache design. Also, the speed disparity between processors and main memory is aggravated. Many research works have been done to
deal with these restrictions and problems. As we know, when the on-chip L2 cache is shared in multi-core architecture, communication problems and evicting interference will be raised among the execution cores. In the multi-core systems, the cache space requirement of each processor and the sharing of the cache among different processor cores should be explored to achieve performance optimization.

In order to achieve the performance optimization, different partitioning schemes and replacement policies for shared level-2 cache were widely used in multi-core architecture. Least Recently Used (LRU) policy as the best replacement policy has been further exploited by Hussein Al-Zoubi [8] to address the shortcomings of LRU. Bloom Filter [1] was also introduced to optimize cache performance. Different cache partitioning techniques have been proposed to improve cache performance, and system performance. However, most of the proposed techniques have poor resource sharing and high overhead in the multi-core systems.

Different application programs have different cache resource requirements, and the performance of each application program can be influenced diversely when the cache space is increased. According to the cache utility criterion, the application programs can be classified into three categories: cache-friendly, low-intensive and cache fitting. The cache-friendly application programs are very sensitive to the changes of cache resource allocations. Comparatively, the low-intensive application programs have a small working set size that can be satisfied by a fixed cache size. For the cache fitting application programs with a huge working set size, comparable cache resource is required for the system to have a good performance.
In this thesis, we propose a Dynamic Cache Partitioning Scheme based on Bloom Filter to address the problem. The proposed cache partitioning scheme will improve both temporal locality and spatial locality of the application programs to take advantage of the memory hierarchy in the multi-core systems. We use multi2Sim, a multi-core simulator, to collect the memory traces for the test benchmarks. The collected memory traces were used as the test inputs for the proposed dynamic cache partitioning scheme. The simulation results showed that the cache misses were reduced significantly by the proposed dynamic Cache Partitioning scheme.

1.3 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 reviews some basic concepts and definitions related to the proposed cache partitioning scheme including memory hierarchy, cache organization, and bloom filter. Chapter 3 proposes and illustrates the Dynamic Cache Partitioning Scheme based on Bloom Filter. Chapter 4 introduces the multi-core simulation environment, Multi2Sim. Multi2Sim is used to collect memory traces for the test benchmarks which will be used as the test inputs for the proposed dynamic cache partitioning scheme. The related work is discussed in Chapter 5. Chapter 6 concludes the thesis.
CAPTER 2

BASIC CONCEPTS ABOUT CACHE PARTITIONING

2.1 Introduction

In this chapter, some basic concepts and definitions related to our proposed cache partitioning scheme will be introduced, including memory hierarchy, cache organization, cache partitioning, and bloom filter. Additionally, hash functions, replacement policies, and Bloom Filter are also illustrated.

2.1 Architecture Overview

The cache organization and cache replacement policy have been intensively studied and analyzed in the uni-processor systems. An architecture overview of an uni-core processor is illustrated in Figure 1.1. In the uni-processor systems, there is an on-chip private cache supporting only one execution core.

Multi-core processors become the major-trend of the new generation of processors recently. A multi-core system presents new challenges for cache hierarchy design, which is much more complicated. The cache organization in the multi-core systems is more complicated. Figure 1.2 shows the architecture overview of a multi-core processor system. Compared with the last generation uni-core processors, the on-chip cache located on the multi-core processors can be shared by many execution cores at the same time. The sharing of the cache between different processor cores present many new challenges for the effective use of the shared cache in the multi-core systems.
Today, two or three levels of on-chip cache are more popular in the multi-core systems. The level one cache with the smallest space is located closest to the execution core, but it has the fastest accessing time. Usually the level one cache is private and divided into two parts, one for data resource and one for instructions. In the cache hierarchy, the next level cache is L2 cache. It has longer accessing time but larger capacity than the level one cache. The last level cache, i.e., level three cache provides the largest cache space, but has the longest access latency. In different multi-core architectures, the level three cache can be located either on-chip or off-chip. In fact, the level three caches are not commonly used in most of the cache partitioning schemes, so the last-level cache typically refers to shared level two cache which is nearest to the off-chip memory but on-chip. It can be shared by all the execution cores in the multi-core systems.
2.2 Memory Hierarchy

Memory performance has a dominating effect on the overall system performance, especially for the data-dominated applications in the computing systems [10]. In practice, the memory system consists of linear arrays of bytes, and during the execution of an application program, the CPU will access memory units that hold the data required in a constant period. Due to the wide gap between CPU and main memory speed, an approach for organizing memory and storage system known as Memory Hierarchy is suggested. The concept memory hierarchy is utilized in computer architecture when system performance issues are evaluated in computer organization design, date movement, algorithm analysis and lower level programming such as locality of reference [10]. In modern memory hierarchy model, a typical computer system provides different levels of access times and capacities of memories.
for data movement. A “Memory Pyramid” is given in Figure 2.1 as an overview of the memory hierarchy.

**Figure 2.1 Memory Pyramid**

As shown in Figure 2.1, memory system is categorized into six levels from M0, the top of the Pyramid, to M5, the end base. In a typical computer system, M0 represents CPU registers, which has the fastest speed but the smallest capacity. The M1 level memory provides larger storage capacity with longer access latency. It is usually an on-chip cache. Then M2 level, is usually an off-chip cache, which transfers data by cache lines from the main memory without any intermediary. The next level, M3, is considered to be the main memory holding disk blocks retrieved from M4. Main memory is well known as the most used storage. The main memory roughly has access time approximate several hundred nanoseconds. And main memory offers bigger capacity at a lower cost. Local disks are located in the M4 that transfer data by
files to the final level. The M5 represents remote secondary storage such as file system and web servers. As the last level memory, M5 has the largest storage capacities but the slowest access time among the whole memory hierarchy.

In fact, the executing programs in a computer system tend to access the data at any memory level more frequently than they access the next lower level memory. Comparatively, the access time at next memory level will be slower, and thus larger and cheaper per byte. The overall effect in the “Memory Pyramid” is that storage capacities are gradually larger from top to bottom of the memory hierarchy, but the data access becomes faster near the top of the memory hierarchy. So the most important point about the memory hierarchy design is the trade-offs between the speed and the size.

In practice, there are two top aims of memory hierarchy development. One is to provide CPU with necessary data as fast as possible and another one is to reduce interference and traffic on the memory bus. To achieve these two goals, cache organization should be the first-order issue that needs to be considered.

### 2.3 Cache Organization

Cache is one of the most significant elements in memory hierarchy and CPU architecture. In computing, two types of locality, spatial locality and temporal locality, are usually exhibited in the program code. Spatial locality suggests that nearby location data will be referenced soon in the near future. And temporal locality suggests that data that has recently used is likely to be access again shortly. Caches in
the computer systems are used to take advantage of the spatial locality and temporal locality of the application programs. As the precise definition from [11] states, “cache is a component that transparently stores data so that future requests for that data can be provided faster.” That is the reason that caches work effectively in CPU architecture.

2.3.1 Cache Hierarchy

In the memory hierarchy, cache is a fast copy of the slower main memory but offering smaller size than that main memories offer, because cache is usually on-chip. It is obvious that the maximum size of the processor chip will be restricted by economic and physical limits. With the advancement of electronic engineering and technology, an increasing number of transistors have been crammed on the processor chip. In this case, the cache sizes grow considerably, but even the largest cache size is tens of megabytes which is still a long way from the gigabytes of main memory or terabytes of disks. The cache consists of small storage units called cache lines (cache blocks, or cache sets) which mirror a segment of the main memory. The size of these units is called the cache line size and holds typically approximate 32 or 64 bytes. Cache can load and store data from the main memory by cache lines.

Caches are commonly organized as level one (L1) cache, level two (L2) cache and level three (L3) cache. The L1 cache has the smallest capacity and is located closest to the execution core, which has the fastest accessing time. Generally the L1 cache is private and further split into data and instruction caches. This split helps
diminish and even avoid pipeline bottlenecks. In addition, the separate caches for instructions also allow for alternate implementations which may take advantage of the instruction streaming. In the cache level hierarchy, the next level cache is L2 cache, which is near L1 cache. It has longer accessing time but larger capacity than L1 cache. The last level cache is level three cache offering the largest capacity and longest access latency. In different processor architectures, the L3 cache can be located either on-chip or off-chip. In practice, the L3 cache is not commonly used in most of the cache partitioning schemes, so the last-level cache typically refers to the shared L2 cache. It is nearest to the off-chip memory but on-chip that can be shared by all the execution cores in the multi-core architectures. Most of the cache partitioning schemes are applied on the shared L2 cache.

2.3.2 Cache Entries

Data is transferred between main memory and cache by cache lines. A cache entry will be generated if a cache line is copied form main memory into the cache. Each entry can be divided into two parts. One part holds the data and the other part stores the part of the memory address, the tag. When the processor wants to reference a memory location in the main memory, the entries in the cache would be traced first. If the processor discovers that the required data exists in the cache, a cache hit would occur. However, if the processor can’t find the required data in the cache, a cache miss has occurred. Typically, the cache miss can be divided into three categories mainly: cold (compulsory) miss, conflict miss and capacity miss. Cold misses will occur when
the cache is empty. Conflict misses occur when the current level cache is large enough, but considerable data objects all map to the same cache line located in the cache. Then capacity miss will arise, if the cache is not big enough to contain all the necessary data accessed.

The proportions of memory accesses that cause a cache hit and a cache miss are respectively defined as cache hit rate and cache miss rate. Read miss will cause execution delay, because the processor has to wait for the necessary data transferred from the main memory. It is much slower to access the main memory than accessing the data from the cache directly. On the other hand, write misses will occur without accessing latency, because the processor can continue working without any stop when the data is transferred to the main memory.

In cache hierarchy, cache misses are easily influenced by multiple factors such as cache size, cache block size and associativity. As shown in Table 2.1, when the cache size is increased without changing other two parameters, the number of potential cold misses will rise, because more entries will be missed in the cache. Increasing the cache block size will lead to fetching more relevant data on each cache miss, so cold misses will be reduced. However, the block size increase will have less influence on the capacity misses. Similarly, both cold and capacity misses can’t be affected by the change of associativity. Comparatively, conflict misses are mostly independent of the cache size, but may go up with the increasing cache block size. And conflict misses are very sensitive to cache associativity.
<table>
<thead>
<tr>
<th>Miss Category</th>
<th>Cache Size</th>
<th>Block Size</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cold</td>
<td>0/-</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Capacity</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Conflict</td>
<td>0</td>
<td>0/-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 2.1 Effects on Miss Types for Increasing Cache Parameters

2.3.3 Cache Associativity

As we know, cache replacement policy will decide the victim cache line to be evicted by the new cache entry. If the replacement policy can select any entry in the cache to hold the required data, the cache will be named fully associative [11]. Contrary to the fully associative cache, if every cache entry in the main memory maps to just one corresponding location in the cache, the cache is called direct mapped cache. If each cache entry in the main memory can map to any one of the N locations in the cache, this cache is N-way set associative. Figure 2.2 shows the mapping schemes for these three types of cache.
Fully associate caches will allow a cache line to take up any cache entry of the cache, which can help avoid aliasing problem, because all the cache entries are open for use. However, hardware overhead will be much higher since every possible cache entry should be traced to decide if it is a cache miss or hit.

Direct mapped caches provide only one entry in the cache to be evicted by each cache line, which has the smallest hardware overhead but the aliasing problem can’t be avoided. Because the same cache line has to be shared by multiple memory locations in the Direct mapped cache.
In fact, the set associative caches are a hybrid of direct and fully associative caches. The cache is divided by ways, and any address can be mapped to any way of a cache line. So the higher associativity cache can hold more possible data from different memory locations and less aliasing problems, which contributes to the overall performance optimization.

2.4 Cache Partitioning

As the multi-core processor is widely used in the computer architecture, more and more cache partitioning technologies acting on the shared level two cache are proposed. Cache Partitioning is to partition the cache among different array variables in a single application program, or partition the cache among different application programs on a uni-core processor, or partition the cache among multiple application programs running on a multi-core system. The result of a cache partitioning scheme is the number and sizes of the cache partition. Cache partitioning can be coarse-grain or fine grain. Way-partitioning [17, 22] divides the cache by ways, so each cache partition is a subset of cache ways. Way partitioning is limited to a few coarse-grain partitions, i.e., the number of cache partitions is less than or equal to the number of ways in the cache. Way partitioning usually reduces the associativity of each cache partition. Cache partitioning schemes partitioning the cache by cache lines instead of cache ways, have also been proposed [6,17], which is also coarse-grain. This type of cache partitioning scheme might need additional hardware or software support, which is costly. Modifying the cache replacement policy [2,17] can also be used to partition
the cache, but this cache partitioning scheme has less control over the cache partition size. Our proposed dynamic cache partitioning scheme based on the bloom filter is the combination of the way partitioning and optimized cache replacement policy.

2.4.1 Hash Functions

Different hash functions are utilized to reduce the cache miss rate. XOR-based mapping function [13] was proposed by Gonzalez et al to reduce conflict cache misses. Then Kharbutli introduced “prime displacement” [14]. In this scheme, additional bits were added as an offset to gain the cache index. In 1997, Bodin et al. proposed a new technology named “skewed associativity” for system performance optimization [12]. In the N-way associative cache, the memory location is hashed by the number of total cache line mod N. However, for the N-way skewed associative cache, different mapping functions are used for each corresponding way. By employing skewed associative scheme, the possible aliasing problems are mitigated because these parallel executed functions allow the processors to reference a different cache line of each way in every cache access.

2.4.2 Replacement Policy

In order to decide the victim cache line for the new entry when a cache miss occurs, different replacement policies are proposed to choose the proper entry to be evicted. The near future prediction for the existing cache entry that least likely to be used is the first order issue for any cache replacement policy. In fact, there is no perfect
replacement policy since the future is too difficult to predict.

As we know, least recently used (LRU) cache replacement policy, replaces the least recently accessed entry, and it is the most frequently used cache replacement policy. A time stamp should be added for each cache entry to track the least recently used cache line. The time stamp of each cache entry will be updated, every time a cache line is accessed. Contrary to LRU, the most recently used (MRU) evicts the most recently used cache entry. In some particular situations, the MRU replacement policy can achieve higher cache hit rate than LRU due to the different application requirements. Hussein Al-Zoubi [8] proposed an approximate LRU’s scheme known as Pseudo LRU (PLRU) that uses one bit per cache line to control the period of reset. In random replacement (RR), the victim cache entry is randomly selected. This cache replacement policy does not need to trace any history information or predict any future activity.

2.5.3 Bloom Filter

A Bloom filter as a probabilistic data structure, providing space-saving, can be used to check whether an element is a member of a set [20]. For the Bloom filter, false positives are accepted, but false negatives are not. Bloom filter acts as a bit array, which can store a big amount of data into a small space. The hash function is applied to a data item to decide the corresponding position of the data item in the bit array. Fig 2.3 shows how a set of numbers x, y and z are stored in a bit array with 18 bits. The bits are numbered from 0-17. After these
designated hash functions are applied on x, y and z to get their corresponding positions, ones are stored in these corresponding positions. For example, suppose h(x) = 1, 5, 13, then ones are stored in position 1, 5, and 13. Now if a data item “w” needs to be tested to decide whether “w” is in the set or not, we can apply hash function on the data item “w” and then check the positions obtained by the hash output within the bit array. If one of the positions has a 0, we can conclude that the data item “w” is not in the set.

Figure 2.5 Bloom Filter Principle
CHAPTER 3

DYNAMIC CACHE PARTITIONING FOR MULTI-CORE SYSTEMS

3.1 Introduction

Different application programs have different cache resource requirements, and the performance of each application program can be influenced diversely when the cache space is increased. In the multi-core systems, sharing the cache among several cores brings new constraints and challenges to cache design. In the multi-core system systems, how to allocate cache resource to satisfy the requirements for each executing processor is the most critical element for performance optimization. In this chapter, we propose and illustrate a dynamic cache partitioning scheme based on bloom filter which can adapt to different cache size requirements and dynamically re-distribute cache allocations for multi-core processors.

3.2 Overview

Konstantinos Nikas, Matthew Horsnell and Jim Garside proposed a dynamic cache partitioning scheme named “Adaptive Bloom Filter Cache Partitioning” in [2], extending the BF mechanism to the multi-core systems. A bloom filter array based on a hash function is used for each cache set to identify every far miss occurred in the shared cache to guide dynamic cache partitioning. Through monitoring the counters of
far cache misses and hits in each cache set, this scheme is able to allow a different partition for each cache set.

In this thesis, we propose and develop a dynamic cache partitioning scheme based on the cache partitioning scheme presented in [2]. We add additional key components (mark array, flag monitor, etc.) and modify the cache monitoring and partitioning module in our proposed dynamic cache partitioning scheme, and the details are illustrated later. Our proposed dynamic cache partitioning scheme combines the way partitioning with updated cache replacement policy. Each cache set will have a different partition of cache ways for the application programs running on the multi-core systems by our proposed dynamic cache partitioning scheme.

Figure 3.1 shows the overview of the system structure of our proposed dynamic cache partitioning scheme based on the bloom filter. For each core, there is a mark array to record the ways of each cache set that are allocated to this particular core. Each bit of the array is mapped to each cache way. When one way is available to be written and read by this core, the corresponding position of the mark array of the core will be set. During the cache partitioning process, mark array will be checked before the cache way is accessed so that every core only references the cache ways of a cache line allocated to this particular core. After every one million clock cycles, all the mark arrays will be updated by the monitoring and partitioning module in the shared last-level cache.
The key component of our proposed dynamic cache partitioning scheme based on the bloom filter is the cache partitioning and monitoring module, which is illustrated in Figure 3.2. As shown in Figure 3.2, the cache monitoring module employed a combination of Bloom filters and two counters to finger out how to recollect the sharing cache resource in each one million clock cycles and redistribute the cache allocations for multi-processors. In our proposed dynamic cache partitioning scheme, the bloom filter is a key element. For the Bloom filter used in our proposed dynamic cache partitioning scheme, false positives are possible, but false negatives are not.
The monitoring module in the dynamic cache partitioning scheme tracks the cache way requirements of each running application and dominates the reallocation of the shared cache resource for each executing core. When a core occupies one cache line in the cache, then the flag monitor will help core ID make a mark on the corresponding field of that cache line which are the first several bits in the tag called flag in Figure 3.2. This core ID recording is necessary for tracking the misses and hits for each core by the two counters and also limits the cache line to be visited by
unauthorized cores. All of the recorded information will be used to re-allocate the shared cache dynamically in each million clock cycles.

### 3.3 Tracing Cache far-misses

Our partitioning module tracks cache misses and hits for each executing core by two counters to guide the reallocation of the cache. However, recording all the cache misses in the shared level cache will be expensive and not necessary. Instead, the cache miss counter focuses on the cache misses that may have the possibility to become cache hits if more cache resource can be provided. This type of cache misses are defined as “far-misses” [2] which are monitored by the Bloom filters and tracked by a far_miss counter shown in Figure 3.2.

In our proposed dynamic cache partitioning scheme based on bloom filter, one Bloom Filter array (BF_array), one far-miss counter and one hit counter are added to each cache set of each processor core. This BF_array should consist of $2^k$ bits which can be mapped by the k least significant bits of one tag in each cache line through the Hash function. In each accessing miss, this part of bits will be hashed to one corresponding position of the BF_array. When a cache miss occurs, the corresponding bit in the BF_array should be set. If a cache miss occurs again in the shared level cache, the k least significant bits of the required tag are looked up in the BF_array by the corresponding entry generated by the Hash function. If the bit has been set in the BF_array, it means that this tag is required in the near past. In this case, far-miss occurs and the counter should be increased. On the other hand, when the BF_array bit
is found to be not set, it means that the requested tag has not been stored in the cache in the close past. On the contrary, when the bit is tested to be set, false positives might have happened, due to the working mechanism of the Bloom Filter. The reason is that multiple different tags might be mapped to the same bit in BF_array. In addition, the amount of true entries in the BFA will not be in accordance with the bits that have been set. In this case, the exactly number of cache ways that should be provided for the core for which the far-misses will be increased. However, considering the significant increase of the overhead of the hardware and the system complexity for dealing with this problem, additional algorithms and changes in the hardware are not preferred. In practice, we decided that the cache allocation of one core should be increased or decreased by one cache way in each re-partitioning.

Cache hits are also tracked through the LRU replacement policy and recorded by the hit counter in our cache monitoring module. Like the BF_array, two counters are used for each cache set to record the far-misses and hits. And in every one million clock cycles, both the two counters and the BFA in each set should be reset.

### 3.4 The Dynamic Partitioning Algorithm

The recorded far-misses and hits recorded in the counters provide important feedback to estimate possible performance gains and losses when the cache allocation is updated for each processor core. The estimated performance losses and gains calculated using the counters are shown below:

\[
\text{lose} = C_{hit} \quad (I)
\]
\[ \text{gain} = a * C_{\text{far}} \quad (2) \]

Here, \( C_{\text{hit}} \) is the cache hit counter for each core. The performance loss is determined by the value of cache hit counter, \( C_{\text{hit}} \), for each core as shown in (1). In (2), the cache far-misses counter, \( C_{\text{far}} \), records the cache far-misses which is otherwise possible to be a cache hit if the shared cache resource for this core is increased. In (2), \( a \) is a crucial factor that is used to adjust the influence of the \( C_{\text{far}} \) counter. As shown in (3), \( a \) can be considered as the percentage of cache ways that weren’t occupied among the total number of cache ways in each set.

\[ a = 1 - \text{occupiedways} / \text{totalways} \quad (3) \]

The pseudo code algorithm for the dynamic cache partitioning based on the bloom filter is presented in Algorithm 3.3.1 and 3.3.2.

In Algorithm 3.3.2 the input of the cache partitioning algorithm is the current allocation of the cache ways recorded from the mark array for each processor core. The output is an updated cache allocation of the cache ways for each processor core. The algorithm reads the cache far miss and hit counters of each set from each processor core. In each iteration, the algorithm uses the function get_occupy() whose pseudo code is shown in Algorithm 3.3.1 to obtain the number of occupied ways for core \( i \) in the cache set \( j \). In the Get Occupy Algorithm, it traverses from the first cache way to the end within set \( j \) to achieve the ways accessed. And then the performance gains and the performance losses from different cores will be stored in two arrays, \( \text{gain}[S] \) and \( \text{lose}[S] \). \( S \) is the number of cache sets. After the second for loop iteration, the maximum value of the estimated performance gain and the minimum value of the
estimated performance loss will be computed for each core. After all the iterations are completed, the algorithm gets the estimated performance gains and the estimated performance loss for each core and then compare these two estimated values. If the former is larger than the later, the core which holds the maximum estimated performance gain value is allocated one more way and the core with the minimum estimated performance loss value is decreased by one way. And this process can be repeated until all the cores are considered, so the time complexity of this algorithm is \( \max(\ O(N*S), \ O(N*\log N) \) ), where \( N \) is the number of cores in the multi-core systems. As a result, the algorithm estimates the cache requirements of each core for each cache set and then updates the cache allocation by one cache way for each processor core.

\[
\text{Algorithm 3.3.1 } \text{Get Occupy Algorithm}
\]

**Input:** The core ID, set ID and the number of sets;
**Output:** The number of ways have been occupied in the set within this core;

\[
N = \text{core\_ID}; \\
S = \text{set\_ID}; \\
S\_\text{num} = \text{set\_number}; \\
\text{Occupy} = 0;
\]

\[
\text{for } \text{way } i = 0 \text{ to } S\_\text{num} \text{ do} \\
\quad \text{Check the core\_ID flag in this cache line} \\
\quad \text{if the valid bit of the core[N].sets[S].way[i] = 1 then} \\
\quad \quad \text{Occupy }++;
\]

\[
\text{end if;}
\]

\[
\text{end for;}
\]

\[
\text{Return Occupy;}
\]

The working mechanism of the algorithm is illustrated as follows. In every one million clock cycles, our cache partitioning algorithm is executed. During this period of time, the amount of cache ways allocated to each processor core will not be
changed, but collection of ways will be updated by one. In execution, the cache ways for two of the executed cores chosen by the algorithm will be reallocated by one way in each iteration. One cache way will be moved from one core to another one which is the recipient to achieve more performance advantages over the old cache allocation. Then the method to choose the two cores whose cache ways will be adjusted by one way should improve the overall system performance.

In addition, the algorithm assigns one authorized way for each core to avoid starvation problems, since some cores may have excessive competitiveness to capture cache resource due to the requirements of its running applications. And all the data stored in the cache lines within the way that is reallocated to another core will be cleared to avoid data interfere among different cores.
The Exploration of The Unassigned Area Scheme

In the algorithm proposed in the last section, all the cache ways are dynamically partitioned and assigned to different cores in every one million clock cycles. In this case, strong isolation is provided in the shared level cache, so more considerations should be given for associativity improvement. To ensure the reasonable isolation and then improve the associativity, the shared level cache is divided into two part: assigned area and unassigned area. In the former part, the dynamic cache partitioning

---

**Algorithm 3.3.2 Cache_Partitioning Algorithm (way[Ncore])**

Input: The original allocation of ways for each core; 
Output: The update allocation of ways for each core; 
N = Ncore; 
S = num_set; 
for core i = 1 to N do 
  for sets j = 1 to S do 
    ways_occupy = get_occupy (core[i], set[j]); 
    gain[j] = a * core[i].sets[j].C_far; 
    lose[j] = core[i].sets[j].C_hit; 
  end for 
  core[i].max_gain = get_max (gain, S); 
  core[i].min_lose = get_min (lose, S); 
end for 
Order all the max_gain and min_lose from min to max 
while max (max_gain) > min (min_lose) do 
  core[max].N_way ++; 
  // core[max] is the core providing the maximum value of max_add 
  core[min].N_way --; 
  // core[min] is the core providing the minimum value of min_lose 
  Ncore = 2; 
  if Ncore <= 1 
    break; 
end if; 
end while 
Return allocation;
scheme is executed normally to allocate reasonable ways for different processor cores. In the unassigned area, cache resource can be used by all the processor cores. In this section, we introduce the unassigned area scheme based on our dynamic partitioning scheme to improve the associativity.

3.5.1 Overview

Figure 3.3 shows the overview of the system structure of unsigned area scheme in the shared cache. In this scheme, the shared level cache is split into assigned area and unassigned area. A Transferring Module is inserted to the assigned area to dominate the flows among these two parts. And the LRU Module added in the unassigned area will help select a victim by LRU replacement policy, when cache miss occurs in the unassigned ways.

In executing, the unassigned ways can’t be accessed directly by any core, since the assigned area has a priority to be visited. The assigned ways will be used by its corresponding core by our dynamic cache partitioning scheme and the misses will be recorded into the Transferring Module. If all the available ways within the same set are occupied and the required data can’t be referenced in the assigned area, the accessing requirement will be delivered to the LRU Module in the unassigned area through the Transferring Module. By this way, this core will be authorized to access the corresponding places in unassigned area. And the LRU Module will provide a proper victim cache line for the core. Then the miss or hit will be sent back from the LRU Module to the Transferring Module.
In the combined cache partitioning scheme, each core owns the full authority of the partitioned ways in the assigned area and also shares all the ways from the unassigned area, so the associativity for each core has been improved greatly.

### 3.5.2 The Unassigned Area Algorithm

In order to maintain the requirement balance between this two area, we set up the constant sizes ratio of 7:1 for assigned ways and unassigned ways respectively. The Algorithm 3.5 shows the scheme in Transferring Module. The inputs of the algorithm are the required tag bits, set ID, the number of ways partitioned in assigned area and the misses sent to the Transferring Module. Comparing the the value of misses with
the allocations of ways, if the value is equal, the required tag should be transferred to the unassigned area. After looking up the corresponding cache lines, the miss or hit will be sent back. On the other hand, if the value of misses is smaller than the number of ways, hit or miss will occur in the assigned part and then transfers back to assigned area.

**Algorithm 3.5  Transfer Algorithm**

**Input:** The tag, set ID, allocation of ways and misses in assigned area  
**Output:** The miss in unassigned area  

1. \( T = \text{tag}; \)  
2. \( S = \text{set}_{\text{ID}}; \)  
3. \( \text{Ways} = \text{allocations}_{\text{way}}; \)  
4. \( M = \text{misses}; \)  
5. \( \text{miss} = 0; \)  
6. \( \text{Hit} = 0; \)  

if \( M = \text{Ways} \) then  
   Transfer the tag to the LRU Module in unassigned area  
   \( \text{miss} = \text{LRU} (T, S); \)  
else  
   \( \text{miss} = 1; \)  
end if;  

Return \( \text{miss}; \)

In fact, although the higher associativity is achieved for each core by the unassigned area scheme, the dynamic partitioning scheme has been restricted and broken seriously. We can see from the Algorithm 3.5, when the conflict misses occur in the assigned area, the core will turn to access the unassigned ways by LRU policy and get the victim cache line in the unassigned area finally. In this case, when all the cache ways was occupied in the assigned area, the dynamic cache partitioning scheme for the unsigned area will not be used. Moreover, the Transferring Module and LRU Module will have more hardware overhead and consume more software resource in
the simulator. Although the dynamic partitioning scheme based on bloom filter was not compatible with the unassigned area scheme well, the combination of these two schemes has a profound significance for our future research.
CHAPTER 4

SIMULATION

4.1 Introduction

We use the Multi2Sim simulation framework developed by Ubal et al. [18] to evaluate our proposed dynamic cache partitioning scheme based on the bloom filter. Multi2Sim is a multi-core simulation framework. We use Multi2Sim to collect memory traces, which will be used as the test inputs for our proposed dynamic cache partitioning scheme. In this chapter, we first introduce the Multi2sim simulation framework and its installation, and then we introduce how to do the performance measurement in Multi2Sim. We also illustrate how to collect memory traces for different benchmark programs running in Multi2Sim. Finally, the simulation results and analysis are presented to show the effectiveness of our proposed dynamic cache partitioning scheme based on the bloom filter.

4.2 Simulation Environment

The overall structure of Multi2Sim simulation framework is presented in Figure 4.1. The Multi2Sim simulation framework consists of four main modules which is implemented in the four main directories named images, sample, tools and src in the Multi2Sim package as shown in Figure 4.1. In the images, considerable icons and
images are included for sample programs and visual tools. The sample consists of test files and programs. And the src is the core part of the simulation package, supports the generation of the executable files by gcc or other compilers. Finally, a large number of libraries that are possible to be used in the simulation are included in the tools. The main advantage of the Multi2Sim simulator is separate functional simulation part and timing simulation part. The users can build up a brand new multi-core architecture using an existing instruction set and programming model by only modifying the timing part with no influence on its functional part. The Multi2Sim source code is organized into various software packages.

In the Multi2Sim simulation framework, the architecture models have been advanced to support compatibility with new multi-core architecture. The four-stage architecture models of the Multi2Sim simulation framework is presented in Figure 4.2. They are a disassembler, a functional simulator, a detailed simulator and a visual tool [19]. These four modules can communicate information with each other and also they work well together without interference.

In the Visual tool model, the User interaction can’t be executed alone without the previous modules. The Functional simulator operates as coordinator to allow the flow of information between Disassembler and the detailed simulator. In the Functional phase, guest programs are supported for execution and then the detailed simulator can track the execution time and the data accessed by the guest programs.
Figure 4.1 Structure of Multi2Sim Source code

Multi2Sim Trunk

images
Icons and other images, mainly used in the M2S-V visual tool.

runtime
Runtime libraries linked with guest code running on top of Multi2Sim. These libraries include OpenCL, CUDA, or OpenGL user-level implementations.

samples
Examples of statically pre-compiled mini-benchmarks, configuration files, and statistic reports. Used in this document as test cases.

src
Multi2Sim C source files.

arch
Implementation of microprocessor architectures. One subdirectory is used for each CPU or GPU architecture supported in Multi2Sim.

x86
Implementation of the x86 CPU architecture. As every architecture, it contains three different subdirectories:

asm
Disassembler.

emu
Functional simulator.

timing
Detailed simulator.

More architectures follow with the same subdirectory structure.

lib
Auxiliary libraries, with implementations for linked lists, hash tables, heaps, and other data structures. Implementation of event-driven simulation modules.

mem-system
Implementation of the memory system, including cache memories, directories, and main memory modules.

network
Interconnection networks, with a model of switches, links, virtual channels, and routing algorithms.

visual
Multi2Sim visualization tool based on GTK.

common
Common files tracking visual state, state checkpoints, and others.

memory
Visual widget representing cache memories and directories.

x86
Visual widget representing the multi-core x86 CPU. For each architecture supported in Multi2Sim, one more directory follows.

tools
Additional tools included in Multi2Sim, probably relying on third-party software, and compiled conditionally. These include an OpenCL kernel compiler and OpenGL shader compiler command-line tools.
4.3 Installation of Multi2Sim

At the web page www.multi2Sim.org, the installation package of the Multi2Sim can be downloaded [18]. We can decompress and unpack the compressed source package of Multi2Sim with the following commands. The right <version> value needs to be used for correct installation.

$ cd multi2Sim-<version>

$ ./configure

$ make

If all the requirements of the libraries are satisfied, we can execute the important command-line tools in multi2sim-<version>/src/m2s after the successful installation of Multi2Sim. The user with System privileges on the machine can select the proper installation package to be available for other users in the system.

The multi2Sim simulator installation command is:

$ sudo make install
Abundant command-line options are provided by the Multi2Sim simulation framework. Users can apply these options to configure the processor settings and system settings. A simple command-line can be used to run the initialization of the simulation environment for a program. The format of the command-line is as follows:

$ ./m2s [options] [program] [args]

In the command-line above, various options are provided by the simulation to specify the performance in <options>. The options can be empty, if they are not necessary. In the command line, <program> represents the name of the executable guest program. Appropriate arguments can be provided for the guest program. A practical example will be given in the following part.

4.4 Memory Trace Collection

To generate the memory access trace of the guest program, some modifications in the Multi2Sim simulator code are needed. The Multi2Sim software package with our modifications needs to be recompiled.

The Multi2Sim source code is categorized into various software packages and displayed in a directory structure as shown in Figure 4.1. There are four main directories under the Multi2Sim. The root directory includes image, sample, tools, and src. In the src directory, an executable file named m2s will be generated after compiling [19]. Firstly, under the main function of m2s.c file, C source code with respect to defining and opening a new file should be added before the functions are launched to collect the memory access trace data. For instance, source code like
“FILE *mtrace” and fopen() function are added to define and open a file respectively. And then we should close the file in the end of main function with function fclose(). Sub-directory mem-system that contains the implementation of the Multi2Sim simulator memory system is also shown under src. In the mem-system, a function named mod_access() in module.c file is modified to record the memory trace data every clock cycle when the memory locations are accessed. The second modification is to add the source code including a fscanf() function to obtain and write the generated memory address data into the file “mtrace” which you created and opened in m2s.c under the src directory. The file name “mtrace” should be defined as a global variable, so that it can be used in different files for timing and data tracing.

After all the modifications are correctly done, the source code should be recompiled in the Multi2Sim simulation framework. The Multi2Sim simulation environment provides more robust and complete support for the x86 instruction set [19]. Through recompiling and launching the simulation environment whose source code has been modified, a new executable file named m2s is generated. The following commands are used to recompile the Multi2Sim simulation framework.

`.configure`

`.make`

The compiled guest program is put in the same directory as the executable file m2s. For instance, 400.perlbench is a computing program. It needs to be launched in the Multi2Sim simulation framework to obtain the memory trace data. Then this guest computing program, 400.perlbench, should be compiled and launched in the
Multi2Sim simulation framework by the following commands:

```
./m2s --x86-sim detailed perlbench_base.i386 attrs.pl
```

In the above command, “./m2s” is the command to launch the Multi2Sim simulation framework. “--x86-sim detailed” are the options for running the guest executable program in the Multi2Sim simulation framework. Multi2Sim provides a set of complete supports for the x86 instruction set, so we can compile and simulate our own source code regardless of gcc, glibc or Linux kernel version. In the command, “perlbench_base.i386” is the guest execution file to be run in the Multi2Sim simulation framework and “attrs.pl” is the default parameter of this guest executable program, perlbench_base.i386. After the guest executable program, perlbench_base.i386, is finished in the Multi2Sim simulation framework, the memory trace data for 400.perlbench has been written into the file “mtrace” created in main function of m2s.c. The memory trace data file “mtrace” will be also generated in the scr directory. Similarly, all the other guest benchmark programs can be launched in the Multi2Sim simulation framework to obtain their memory trace data. These test benchmark programs used as test cases for our proposed dynamic cache partitioning scheme based on the bloom filter are 401.bzip2, 429.mcf, 437.leslie3d.ref, 433.milc, 471.omnetpp, 473.astar.ref, 450.soplex, 462.libquantum, 470.lbm, and 455.gobmk, which will be introduced in the following sections.

### 4.5 Simulation Environment

In our experiments, we use the Multi2Sim simulation framework to obtain the
memory trace data for the test program benchmarks. We construct a two-level cache simulator for the multi-core systems whose inputs are the memory trace data. We evaluate our proposed dynamic cache partitioning scheme based on the bloom filter for a multi-core system with a shared L2 cache. Table 4.1 below shows the parameters of the basic cache simulator configuration.

<table>
<thead>
<tr>
<th>Number of Processor cores</th>
<th>4 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 caches</td>
<td></td>
</tr>
<tr>
<td>Private instruction and data caches</td>
<td></td>
</tr>
<tr>
<td>32KB, 4-way set associative, 128B line size</td>
<td></td>
</tr>
<tr>
<td>Level 2 caches</td>
<td></td>
</tr>
<tr>
<td>Shared instruction and data caches</td>
<td></td>
</tr>
<tr>
<td>256KB, 32-way set associative, 128B line size</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1 Parameters of The Cache Simulator Configuration.

As shown in Table 4.1, the cache simulator supports a quad-core system architecture. In the cache simulator, a 256KB, 32-way shared level two cache and a 32KB, 4-way private level one cache are provided.

For each evaluation, we use a mixed set of the memory trace data of four different test benchmarks generated from the Multi2Sim simulation framework as the inputs for the dynamic cache partitioning scheme based on the bloom filter to test the effectiveness of our cache partitioning scheme for a quad-core. The four mixed sets of four different benchmarks are given in the Table 4.2.
Table 4.2 Combinations of Benchmarks

<table>
<thead>
<tr>
<th>Mix01</th>
<th>400.perlbench, 454.calculix, 445.gobmk, 437.leslie3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mix02</td>
<td>473.astar, 470.lbm, 471.omnetpp, 437.leslie3d</td>
</tr>
<tr>
<td>Mix03</td>
<td>435.gromacs, 416.gamess, 450.soplex, 483.xalancbmk</td>
</tr>
<tr>
<td>Mix04</td>
<td>401.bzip2, 437.leslie3d, 450.soplex, 473.leslie3d</td>
</tr>
</tbody>
</table>

According to the cache utility criterion, applications can be classified into three categories: cache-sensitive, low-intensive and cache-fitting. The cache-sensitive application programs are very sensitive to the changes of cache resource allocations. Comparatively, the low-intensive application programs have a small working set size that can be satisfied by a fixed cache size. For the cache fitting application programs with a huge working set size, comparable cache resource is required for the system to achieve a good performance. The cache-sensitive test benchmark programs include 401.bzip, 416.gamess, 437.leslie3d, 445.gobmk, 470.lbm and 473.astar. The cache-intensive test benchmark programs include 400.perlbench, 450.soplex, 471.omnetpp and 483.xalancbmk. And there are three cache-fitting test benchmark programs named 429.mcf, 435.gromacs. Each mixed set of the test benchmarks includes 4 various carefully chosen test benchmarks with different memory access patterns, i.e., each mixed set of the test benchmarks include cache-sensitive, low-intensive and cache-fitting benchmark programs.

4.6 Simulation Results and Analysis

The proposed dynamic cache partitioning scheme based on the bloom filter was tested
in the multi-core systems using all the test benchmarks. The simulation results and the performance analysis are then presented. The testing benchmarks are classified into four mixed benchmark sets, Mix01, Mix02, Mix03, Mix04, as the test inputs for our proposed dynamic cache partitioning scheme in the quad-core systems. Figure 4.3 presents the cache performance of our proposed dynamic cache partitioning scheme compared with a naïve cache using the basic LRU for the test benchmark set Mix01 as shown in Table 4.2.

In Figure 4.3, the Y-axis indicates the cache miss rate and the X-axis shows the name of the benchmarks. For each benchmark, there are two corresponding bars in the graph. The blue bar represents the cache miss result for the cache with the basic LRU policy and the purple bar represents the cache miss result by our proposed dynamic cache partitioning scheme based on the bloom filter. Figure 4.3 shows that our proposed dynamic cache partitioning scheme based on the bloom filter causes 0.04% increase of cache miss rate on the processor core that runs the 454.calculix benchmark, however, nearly 0.5% and 0.2% reduction of cache miss rate is achieved for the processor cores running 400.perlbench and 445.gonmk. And the core running the benchmark 437.leslie3d is neither increased, nor decreased. As the cache-intensive benchmark in the mix01 set as shown in Table 4.2, 400.perlbench can be satisfied by a small fixed cache size. However, the requirement of benchmarks 454.calculix will be difficult to be fulfilled as a cache-fitting type. The cache performance results of Mix01 set shown in Figure 4.3 shows that the proposed dynamic cache partitioning scheme based on the bloom filter allocated a proper cache size for the 400.perlbench
and then provided relatively adequate cache resource to the cache-sensitive benchmarks (437.leslie3d and 445.gobmk) but limited the cache size for the 454.calculix. On average, the proposed cache partitioning scheme reduced the cache miss rate for 0.17%, so it improved the overall system performance.

![Figure 4.3 Results of Mix01 by our cache partitioning scheme in a quad-core system](image)

Figure 4.3 Results of Mix01 by our cache partitioning scheme in a quad-core system

Figure 4.4 presents the cache performance of our proposed dynamic cache partitioning scheme based on the bloom filter compared with a naïve cache using the basic LRU replacement policy for the test benchmark set Mix02 as shown in Table 4.2.

As shown in Figure 4.4, our proposed dynamic cache partitioning scheme based on the bloom filter reduces the cache miss rate for the processor core running the test benchmark 471.omnetpp by 0.41%, the core running the test benchmark 473.astar increases the cache miss rate by 0.1%, which is not good. And the cache miss rates of the processor cores running 437.leslie3d and 470.lbm keep are neither increased, nor
decreased. In the Mix02 set, 471.omnetpp is a cache-intensive benchmark and the remaining test benchmarks are all cache-sensitive programs. The results show that the proposed cache partitioning scheme allocated a proper cache size for the benchmark 471.omnetpp and then provided relatively adequate resource to two of the cache-sensitive benchmarks. On average, the proposed cache partitioning scheme still improved the overall system performance.

Figure 4.4 Results of Mix02 by our cache partitioning scheme in a quad-core system

Figure 4.5 and Figure 4.6 present the cache performance of our proposed dynamic cache partitioning scheme based on the bloom filter compared with a naïve cache using the basic LRU replacement policy for the test benchmark set Mix03 and Mix 04 respectively.

In Figure 4.5, the processor core running 483.xalancbmk achieved the largest reduction of cache miss rate, 1.0% (1.05%), and a small reduction of cache miss rate, 0.02% was obtained for the processor core running the benchmark 416.gamess. The
cache performance of the processor core running 450.soplex was neither degraded, nor improved. Unfortunately, the cache miss rate of the processor core running the benchmark 435.gromacs was increased by 0.1%. In the Mix03 set, 483.xalancbmk and 450.soplex are cache-intensive benchmarks. The 435.gromacs is the cache-fitting program. On average, the proposed cache partitioning scheme reduced the miss rate for nearly 0.30%, correspondingly improved the overall system performance. Similar performance analysis can be done for the test results shown in Figure 4.6.

From the figures, we can see that in a quad-core system, one executing processor core may be selected to sacrifice cache resource for greater performance gains of the overall system in each one million clock cycles. The cache performance of the cache-sensitive and cache-intensive benchmarks can be improved by our proposed dynamic cache partitioning scheme based on the bloom filter. The proposed dynamic cache partitioning scheme adapts to the cache requirement of each benchmark program running on a processor core in every one million clock cycles and then allocates the appropriate cache size for each benchmark program. So unnecessary entry look-ups are avoided as well as cold misses, especially for the cache-intensive benchmark programs that need a small fixed cache resource. For the cache-fitting benchmarks, the required cache sizes will be larger than the available resource, so it is difficult for our proposed dynamic cache partitioning scheme to achieve obvious performance improvement without increasing the cache size. In summary, our proposed dynamic cache partitioning scheme improved the cache performance for the quad-core system for all the four mixed set of benchmarks.
Figure 4.5 Results of Mix03 by our cache partitioning scheme in a quad-core system

Figure 4.6 Results of Mix04 by our cache partitioning scheme in a quad-core system
CHAPTER 5

RELATED WORK

5.1 Introduction

In this chapter, different cache partitioning schemes for shared level caches related to our thesis research will be discussed. The related work on improving the overall performance through optimizing replacement policies are presented first. Then, the last level cache partitioning techniques to reduce miss rate are presented. Next, several cache partitioning techniques for energy saving are discussed. Finally, other cache partitioning approaches addressing different issues in the multi-core systems are introduced.

5.2 Replacement Policies

With the increasing number of cores in the system architecture, it tends to be more difficult to figure out the optimum victim for replacement in cache. Least Recently Used (LRU) replacement policy is the most frequently used cache replacement policy. An excellent replacement policy should improve the hit rate by selecting the right victim that will not be accessed in the near future. However, in the multi-core systems, the baseline LRU replacement policy can’t provide more benefits when multiple cores demand shared cache resources and interfere with each other. Several replacement
schemes have been proposed to address the shortcomings of LRU. Hussein Al-Zoubi [8] proposed an approximate LRU’s scheme known as Pseudo LRU (PLRU) that uses one bit per cache line to control the period of reset. Quereshi et al [9] presented an adaptive replacement policy for which the adaptions are derived from workload combinations running on different processor cores.

5.3 Degrading The Miss Rate

The obvious merit of direct mapped caches is that a faster hit time will be attained compared with the set-associative cache. The drawback is that there will be more conflict misses compared with the set-associative cache due to the strong competitions for the same cache line among multiple memory addresses. Furthermore, the large number of potential data items that will be accessed in the near future will be thrashed out by the data items required by other running applications in the multi-core systems. In order to resolve these issues, several techniques have been proposed.

The concept of Bloom Filter proposed in [1] by Peir er al. Bloom Filter is used as a probabilistic mechanism to maintain the membership for each set quickly. Two types of BF were presented to identify the cache misses: one is the partitioned-address bloom filter and another one is the partial-address bloom filter. For the bloom filter based on partial-address matching, the least-significant bits of the tag bits are used to index a small array of bits named Bloom Filter Array (BFA). Each bit in BFA indicates whether the corresponding match of partial address exists in the cache. In this paper, BF is elaborately designed and sized to reach a precise prediction for the
shared cache misses.

Konstantinos Nikas, Matthew Horsnell and Jim Garside proposed a dynamic cache partitioning scheme named “Adaptive Bloom Filter Cache Partitioning” in [2], extending the BF mechanism to the multi-core systems. A bloom filter array based on a hash function is used for each cache set to identify every far miss happened in the shared cache to guide dynamic cache partitioning. Through monitoring the counters of far cache misses and hits in each cache set, this scheme is able to allow a different partition for each cache set. In this case, better timing performance and lower cost are provided for the increasing number of cores in the multi-core systems.

5.4 Low Energy Cache

The energy consumption includes the dynamic power consumption and the static power consumption. Energy dissipation in caches in the multi-core systems has become one of the most important issues in the research community. Many techniques have been proposed to improve the energy efficiency of the cache.

In [3], Weixun Wang, Prabhat Mishra and Sanjay Ranka proposed a dynamic cache reconfiguration (DCR) technique to reduce the energy consumption of the multicore systems. Through comparative analysis of three approaches: CP (L2 cache partitioning only), DCR + UCP (Reconfiguration of L1 cache partitioning with uniform L2 cache partitioning), and the proposed DCR + CP (reconfiguration of private caches and partitioning of the shared cache), they concluded that integrating L1 DCR and L2 CP would reduce interference between different tasks more
effectively and further enhance the energy efficiency for the real-time tasks.

The energy efficient cache architectures were discussed in [4] for uni-core, multi-core and many-core processors. The authors proposed a cooperative energy-efficient cache partitioning scheme for last level shared cache in the multi-core systems. A way-aligned scheme was used to help reduce both dynamic and static power consumption in this cache partitioning scheme.

5.5 Other Cache Partitioning Approaches

Operating System [5] was used to manage the shared cache in the multi-core systems. In [6], an innovative page placement algorithms called careful-mapping was proposed to reduce cache contention by R. E. Kessler and Mark D. Chenjie Yu and Peter Petrov proposed a cache partitioning scheme considering the widening gap between the memory requirement and limited off-chip memory bandwidth as the most critical issue.
CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis, we proposed a Dynamic Cache Partitioning Scheme based on Bloom Filter to improve the performance of the computing systems. The proposed cache partitioning scheme will improve both temporal locality and spatial locality of the application programs to take advantage of the memory hierarchy in the multi-core systems. We use multi2Sim, a multi-core simulator, to collect the memory traces for the test benchmarks. The collected memory traces were used as the test inputs for the proposed dynamic cache partitioning scheme. The simulation results for the proposed dynamic cache partitioning scheme showed that the cache misses were reduced significantly by the proposed dynamic Cache Partitioning scheme.

This thesis can be extended in the following directions:

In our future work, we will use data dependency analysis, memory access pattern analysis, and memory size estimation to guide cache partitioning. We will also develop hardware-software managed cache partitioning schemes supported by specific cache control instructions.

In addition, software managed Scratch Pad Memories (SPM) reduce hardware management overhead by relying on the compiler to copy code segment and data into the SPM. A recent study [21] showed that the SPM memory has 40% lower power.
consumption than a cache of equivalent size. In our future work, we will explore reconfigurable memory modules in the multi-core systems. The memory size and the data locality analysis will be used to decide whether each memory module should be configured as software controlled Scratch Pad Memory or Cache. We will also study compiler-directed dynamic allocation of SPM to identify most frequently accessed data to be placed in the SPM to reduce the off-chip memory accesses.
REFERENCES


[18] Rafael Ubal , Byunghyun Jang , Perhaad Mistry , Dana Schaa , David Kaeli, Multi2Sim: a simulation framework for CPU-GPU computing, Proceedings of the 21st international conference on Parallel architectures and compilation techniques, September 19-23, 2012, Minneapolis, Minnesota, USA.

[19] Zhongliang Chen, Tahir Diop. The Multi2Sim Simulation Framework: A CPU-GPU Model for Heterogeneous Computing (For Multi2Sim v. 4.0.1). Documentation.

